Simple Parallel Biconnectivity Algorithms for Multicore Platforms

George M. Slota and Kamesh Madduri
Department of Computer Science and Engineering
The Pennsylvania State University
University Park, PA, USA
gslota@psu.edu, madduri@cse.psu.edu

***Abstract*—We present two new algorithms for finding the biconnected components of a large undirected sparse graph. The first algorithm is based on identifying articulation points and labeling edges using multiple connectivity queries, and the second approach uses the color propagation technique to decompose the graph. Both methods use a breadth-first spanning tree and some auxiliary information computed during Breadth-First Search (BFS). These methods are simpler than the Tarjan-Vishkin PRAM algorithm for biconnectivity and do not require Euler tour computation or any auxiliary graph construction. We identify steps in these algorithms that can be parallelized in a shared-memory environment and develop tuned OpenMP implementations. Using a collection of large-scale real-world graph instances, we show that these methods outperform the state-of-the-art Cong-Bader biconnected components implementation, which is based on the Tarjan-Vishkin algorithm. We achieve up to $7.1\times$ and $4.2\times$ parallel speedup over the serial Hopcroft-Tarjan and parallel Cong-Bader algorithms, respectively, on a 16-core Intel Sandy Bridge system. For some graph instances, due to the fast BFS-based preprocessing step, the single-threaded implementation of our first algorithm is faster than the serial Hopcroft-Tarjan algorithm.**

I. Introduction

The biconnected component (BiCC) decomposition of an undirected graph refers to determining all *maximal biconnected subgraphs* or *blocks* contained within the graph. A biconnected subgraph is a graph which remains connected with the removal of any single vertex and all edges incident on it. *Articulation vertices* or *articulation points* are vertices that, when removed, disconnect the graph into multiple connected components. These vertices belong to two or more biconnected components. Finding the articulation vertices in a graph is one of main purposes of BiCC decomposition, as these vertices represent links that are critical for overall connectivity. A *bridge* is defined as an edge that, when removed, would disconnect the graph into multiple connected components. BiCC decomposition also gives us a disjoint partitioning of all edges. Each edge in the graph can belong to only one maximal biconnected subgraph.

Identifying large and non-trivial biconnected components, articulation vertices, and bridges are useful in the analysis and characterization of new graph data. In the field of networking, when designing communication networks and physical infrastructure networks, identifying articulation points and minimizing bridges is relevant to network robustness and redundancy. In large virtual networks such as social networks and web crawls, BiCC decomposition gives insight into network structure and has potential to be a useful preprocessing step in data analysis [26], [27].

A. Contributions

This paper introduces two new shared-memory parallel approaches for finding the biconnected components of large sparse graphs. Both approaches use a breadth-first spanning tree. The first method is based on executing multiple truncated breadth-first searches (BFSes). We demonstrated the use of this strategy to find articulation points in previous work [29]. Here, we extend the method for full BiCC decomposition. The second method uses the *color propagation* [25] technique. We used this strategy in prior work for detecting strongly connected components in parallel. Similar to previous serial and parallel algorithms, the output of both our algorithms is a labeling of edges into disjoint biconnected components and a classification of vertices into articulation and non-articulation points.

We analyze and implement both algorithms and perform an experimental study on an Intel multicore platform. Both approaches demonstrate good parallel strong scaling across a wide range of real-world and synthetic test cases, with the truncated BFS-based approach (the first method) offering the best speedups relative to a serial implementation. Additionally, due to the fact that the first method uses the hybrid direction-optimizing BFS algorithm of Beamer et al. [7] as a subroutine, a substantial fraction of the graph edges are pruned or untouched in some cases. This results in the single-threaded performance of our new approach being faster than the linear-work Hopcroft-Tarjan DFS-based algorithm.

II. Background

There is substantial prior work on serial and parallel algorithms for BiCC decomposition. We review three algorithms that we use for comparison and discuss relevant implementation details in this section.

A. Hopcroft-Tarjan Algorithm

The serial algorithm for BiCC decomposition, designed by Hopcroft and Tarjan [16], is optimal in the RAM model and based on a single Depth-First search (DFS). It runs in $O(n+$

m) time (n is the number of vertices and m the number of edges) and linear space. A recursive implementation to identify articulation points is given in Algorithms 1 and 2.

Algorithm 1 Hopcroft-Tarjan biconnectivity algorithm to identify articulation points.

```
1: procedure HT(G(V, E))
2:     for all v ∈ V do
3:         preorder(v) ← −1
4:         low(v) ← −1
5:         articulation(v) ← false
6:     global count ← 0
7:     root ←selectRandomRoot(V)
8:     HT-DFS(G, root, root)
```

Algorithm 2 Recursive DFS used in Hopcroft-Tarjan algorithm.

```
1: procedure HT-DFS(G, u, v)
2:     children ← 0
3:     preorder(v) ← count++
4:     low(v) ← preorder(v)
5:     for all ⟨w, v⟩ ∈ E do
6:         if preorder(w) = −1 then
7:             children ← children + 1
8:             HT-DFS(G, v, w)
9:             low(v) ←min(low(v), low(w))
10:            if low(w) ≥ preorder(v) and u ≠ v then
11:                articulation(v) ← true
12:            else if w ≠ u then
13:                low(v) ←min(low(v), preorder(w))
```

The algorithm maintains two arrays of size n in addition to the standard DFS stack of visited vertices. One array, the $preorder$, records the order in which vertices are discovered in the DFS. The second array low tracks the lowest DFS depth of the adjacencies or children of the current vertex. When there are no more adjacencies remaining to be explored from the vertex on top of the stack, this vertex is removed. If this vertex does not connect to any vertices lower on the stack other than the vertex immediately preceding it on the stack, then we know that this vertex is an articulation point and can mark it as such.

The listing is only for identifying articulation points. If we would like to partition edges into components, we need to maintain an additional stack. As each edge is first touched, it is placed on this stack. When an articulation point is identified, all of the edges contained in the biconnected component can then be removed from the stack and appropriately labeled. Our serial implementation is closely based on this recursive DFS-based algorithm. We will refer to this as the HT method.

B. Tarjan-Vishkin Parallel Algorithm

The PRAM BiCC algorithm by Tarjan and Vishkin [30] requires $O(\log n)$ time using $O(n+m)$ processors. An overview of the main steps is given by Algorithm 3.

The Tarjan-Vishkin algorithm has several key subroutines. First, a spanning tree T is created from the input graph G. This can be computed using any traversal. T is then rooted at an arbitrary vertex and a Euler tour is found to create an ordered list of vertices L. List ranking is then performed on L, which in turn gives the preordering numbering of vertices

Algorithm 3 Tarjan-Vishkin algorithm to identify articulation points.

```
1: procedure TV-BiCC(G(V, E))
2:     T ← SpanningTree(G)
3:     L ← EulerTour(T)
4:     pre, size ← ListRank(L)
5:     low, high ← DetermineMinMaxPreorder(G, T, pre, size)
6:     G′ ← BuildAux(G, T, low, high)
7:     C ← ConnectedComponents(G′)
8:     B ← BiconnectedComponents(C, G)
```

$pre(v)$ for all $v \in T$. The size of each subtree rooted at v in T, $size(v)$, is also found.

Similar to the serial algorithm, two values are then obtained for each vertex $v \in G$, a low and a $high$ value. Using the preorder numbering from the previous step, we determine the low and $high$ values as the lowest and highest preorder numbering of all descendants of or neighbors of descendants of v that are not connected in T. The next steps utilize these labelings to build an auxiliary graph G', whose connected components C are the biconnected components of G. The problem of finding biconnected components in the original graph is recast as finding connected components in this auxiliary graph. A final step can then be performed to create an explicit set B of the edges and articulation points defining each biconnected component in G.

1) Cong-Bader TV-Filter Algorithm: An experimental study by Cong and Bader [11] presents an improvement to the Tarjan-Vishkin algorithm that leads to a significant reduction in the size of the auxiliary graph. They suggest a preprocessing step that filters out certain *non-essential* edges (i.e., edges that do not impact the biconnectivity of G). The use of this preprocessing step also reduces execution time of other subroutines in the Tarjan-Vishkin algorithm for the graph instances studied in [11]. We also independently notice up to a 4× speedup over the TV algorithm, and hence we just focus on TV-Filter in this paper.

Note that this preprocessing step is similar to the approach for finding k-connected subgraphs in a j-connected graph, where $j > k$, as described by Nagamochi and Ibaraki [24]. However, it is important to note that the filtering step finds only a k-connected subgraph in a k-connected graph, specifically where $k = 1$, by constructing $k + 1$ spanning forests (in this case, a spanning tree and then a spanning forest of the graph with that spanning tree removed). As a result, this preprocessing step preserves the biconnectivity of the original graph by maintaining articulation points, bridges, and the biconnected components themselves.

Algorithm 4 Cong-Bader algorithm to identify articulation points.

```
1: procedure CB-BiCC(G(V, E))
2:     T, P ← BFS(G)
3:     F ← SpanningForest(G \ T)
4:     B ← TV-BiCC(F ∪ T)
5:     for all e = ⟨u, v⟩ ∈ G − (F ∪ T) do
6:         label e as in BiCC containing v and P(v)
7:         B ← (B ∪ e)
```

Algorithm 4 provides an overview of Cong and Bader's approach. A BFS is first performed from an arbitrary vertex to obtain T (the spanning tree) and P (parent information). Edges in T are then filtered from G. A spanning forest F is obtained using another traversal and by executing connected components. The union of edges in F and T is shown to contain the essential edges needed to determine the biconnected components B using the standard Tarjan-Vishkin algorithm. The rest of the edges are non-essential. Once the Tarjan-Vishkin algorithm completes, the non-essential edges are labeled and added back to obtain the complete BiCC output. Correctness proofs and the data structures used to store the intermediate results are given in [11].

In this work, we use an updated version of the code first developed by Cong and Bader for their study [10], which was designed using the SIMPLE POSIX threads-based framework [4]. We changed their code for execution on our test platform, primarily modifying storage of common structures and eliminating or globalizing some thread-owned structures in an effort to reduce memory utilization. Despite these changes, memory usage when creating the auxiliary graph limits running the code on our two largest test instances.

C. Related Work

There are several other known parallel algorithms for BiCC. One of the earliest CREW PRAM parallel algorithms was presented by Eckstein [13]. This approach runs in $O(d \log^2 n)$ time with $O((n+m)/d)$ processors on the CREW PRAM model. This work was the first to note that the structure of BFS trees can be utilized to find articulation points, and our work can be considered a related extension of this work for modern architectures. Savage and JáJá [28] designed two PRAM algorithms, taking $O(\log^2 n)$ and $O(\log^2 n \log k)$ time and requiring $O(n^3/\log n)$ and $O(mn + n^2 \log n)$ processors, respectively, where k is the number of biconnected components. A CREW PRAM algorithm by Tsin and Chin [31] runs in $O(\log^2 n)$ time with $O(n^2/\log^2 n)$ processors.

Another class of proposed biconnectivity algorithms utilize open ear decompositions. An original approach was described by Maon et al. [22]. This approach was expanded upon by Miller and Ramachandran [23], with a related implementation for solving 2-edge connectivity problems on an early massively parallel MasPar system [17].

In terms of parallel implementations, the Tarjan-Vishkin approach has received the most attention, with Cong and Bader being the first to demonstrate parallel speedup on an SMP system over the serial HT algorithm with their TV-Filter algorithm. Edwards and Vishkin [14] also implemented the TV and HT algorithms using a new programming model, and demonstrated scalability and parallel speedup on the Explicit Multi-Threading (XMT) manycore computing platform.

Most recently, Ausiello et al. developed a MapReduce-based biconnected components detection algorithm [2], [1] under a streaming data model. This method is based on previous work by Westbrook and Tarjan [32], but utilizes a lightweight *navigational sketch* of the input graph to hold biconnectivity information for the full graph.

III. New Parallel Algorithms

We present two new methods in this section that are both based on a breadth-first spanning tree. However, unlike TV and TV-Filter, we do not construct an auxiliary graph. Instead, we identify some properties that articulation vertices must satisfy and use them to decompose the graph. Our two methods use BFS and color propagation as underlying subroutines, and so we refer to them as BiCC-BFS and BiCC-Coloring. The output obtained is identical to prior algorithms. In both the methods, we initially assign two integer labels to each vertex and progressively update them. The final vertex labels of all vertices can be inspected to determine the component to which an edge belongs to. These algorithms assume that we begin with an undirected graph with a single connected component.

A. BFS-based BiCC method

We first describe the BFS-based approach to identify articulation points, similar to that used by Eckstein [13]. The key steps are listed in Algorithms 5 and 6. Assume that G has only one connected component with no multi-edges and self loops. Choose an arbitrary vertex r, designate it as the root, and perform a BFS. Store the BFS output in two arrays, P and L. For all $v \in V$, $P(v)$ stores a parent of v (i.e, the BFS tree) and $L(v)$ gives the depth of v in the BFS tree, or the distance of v from r. For all $v \in V$, we define a *child* w of v to be any adjacency of v such that $P(w) = v$. Thus, the set of all children of a vertex is a subset of its adjacencies. Our algorithm is based on the following proposition.

Proposition III.1. *A non-root vertex v in the BFS tree $\langle P, L \rangle$ is an articulation vertex if and only if it has at least one child w that cannot reach any vertex of depth at least $L(v)$ when v is removed from G.*

The above statement is equivalent to the following.

Proposition III.2. *A non-root vertex v in the BFS tree $\langle P, L \rangle$ is not an articulation vertex if and only if all its children w in the BFS tree ($P(w) = v$) can reach all other vertices in the graph G when v is removed from G.*

Proof: An articulation vertex, by definition, is a vertex whose removal will decompose G into two or more connected components. Consider a vertex v and all its adjacencies. The BFS output splits the adjacencies of any non-root vertex into three disjoint subsets: its children (as defined above), non-child adjacencies at depth $L(v)+1$, and adjacencies at depth $L(v)$ or $L(v)-1$. If v were an articulation point, then the adjacencies of v would be split up such that at least two of them lie in different connected components upon removal of v. If v is not an articulation point, there must be an alternative path in the graph from adjacencies of v to every other vertex, and thus, between every pair of adjacencies of v as well. To show that v is an articulation point, it suffices to inspect just the children of v in the BFS tree and show that at least one of

them cannot reach an alternate vertex at the same level as v (and thus is disconnected from some part of the graph). Non-child adjacencies of v have a path through their parent in the BFS tree to other vertices, and so we do not need to consider them explicitly. If a child can reach some vertex at the same level as v, then it can reach all other vertices by tracing a path back to the root. ■

As a more general extension of the above propositions, we have the following Corollary.

Corollary III.3. *If a traversal from any $u_i \in V$ ($P(u_i) = v$) is not able to reach* all *other $u_j \in G$ ($P(u_j) = v$) when v is removed from the graph, then v is an articulation point. Further, if the only path in G between u_i and u_j requires v, then u_i and u_j are in separate biconnected components with v as an articulation point. We term v as the* parent articulation vertex.

Algorithm 5 BFS-based algorithm to identify articulation points in BiCC decomposition.

```
1:  procedure BFS-ArtPts(G(V, E))
2:      for all v ∈ V do
3:          Art(v) ← false
4:          visited(v) ← false
5:      Select a root vertex r
6:      P, L ← BFS(G, r)
7:      for all u(≠ r) ∈ V where P(u) ≠ r do
8:          v ← P(u)
9:          if Art(v) = false then
10:             l ←BFS-L(G, L, v, u, visited)
11:             if l ≥ L(v) then
12:                 Art(v) ← true
13:     Check if r is an articulation point
```

Algorithm 6 Truncated BFS subroutine in the BFS-ArtPts algorithm.

```
1:  procedure BFS-L(G(V, E), L, v, u, visited)
2:      Insert u into Q
3:      Insert u, v into S
4:      visited(u) ← true
5:      visited(v) ← true
6:      while Q ≠ ∅ do
7:          for all x ∈ Q do
8:              Remove x from Q
9:              for all ⟨w, x⟩ ∈ E where visited(w) = false do
10:                 if L(w) < L(u) then
11:                     for all s ∈ S do visited(s) ← false
12:                     return L(w)
13:                 else
14:                     Insert w into Q
15:                     visited(w) ← true
16:     return L(u)
```

We now describe the algorithm in more detail. Parallelization of the initial BFS computation required to construct P and L is well-studied. We use the parallel BFS from our prior work [29]. This implementation maintains a bit vector for tracking visited vertices, thread-local queues, and further utilizes a direction-optimizing search [7]. These optimizations have been demonstrated to considerably speed up parallel BFS computations on the small-world graphs we are considering.

To identify the articulation points, we consider every vertex u and its parent $v = P(u)$. Instead of performing pairwise reachability queries from u, we perform a BFS from u after removing v from G and track the level of vertices that are reached, using the previously computed L values. If any vertex w with level $L(v)$ or less is reached, we can exit. This step (i.e., step 7 of Algorithm 5) can also be parallelized, with each thread maintaining a separate $visited$ bit vector. We also use another temporary stack S in the BFS-L subroutine to track visited vertices.

The root vertex r must be handled separately. There are several ways to determine whether it's an articulation point. The simplest way is to select a vertex that is definitely not an articulation point (vertex of degree 1), or is certainly an articulation point (a degree-2 vertex, with one of its neighbors having a degree of one). If no such vertices exist, then there are two options. One is to create a new spanning tree using an alternate root and run the second stage of the algorithm with any children u of the original root r, $P(u) = r$. Because our optimized BFS subroutine is quite fast, this is not entirely impractical. The second option is to run a BFS on $G \backslash r$ from a single u where $P(u) = r$ and track whether all other w where $P(w) = r$ are also reachable. If they are all reachable from u, then r is not an articulation vertex as per Corollary III.3.

1) Identifying Biconnected Subgraphs: We now extend the previous algorithm and make it work-efficient to label edges. The new method is given in Algorithms 7 and 8.

Algorithm 7 BFS-based algorithm to perform BiCC decomposition.

```
1:  procedure BFS-BiCC(G(V, E))
2:      for all v ∈ V do
3:          Art(v) ← false
4:          visited(v) ← false
5:          Low(v) ← v
6:          Par(v) ← v
7:      Select a root vertex r
8:      P, L, LQ ← BFS(G, r)
9:      for all Q_i ∈ LQ_{m···1} do
10:         for all u ∈ Q_i, where Par(u) = u do
11:             Remove u from Q_i
12:             v ← P(u)
13:             l, vid_low, V_u ←BFS-LV(G, L, v, u, visited)
14:             if l ≥ L(u) then
15:                 Art(v) ← true
16:                 visited(v) ← false
17:                 for all w ∈ V_u do
18:                     Low(w) ← vid_low
19:                     Par(w) ← v
20:                     visited(w) ← false
21:                     Remove w from V
22:     for all e = ⟨u, v⟩ ∈ E do
23:         if Low(v) = Low(u) or Par(u) = v then
24:             BiCC(e) ← low(u)
25:         else
26:             BiCC(e) ← low(v)
```

The primary goal of this approach is to determine for all $v \in V$ two labels, $Par(v)$ and $Low(v)$. The Par value is the highest-level articulation point (parent articulation vertex) separating v from the root. Low signifies the lowest-value vertex identifier (vertices are numbered from 0 to $n-1$) among all vertices contained in the biconnected component of v. We can then use these two vertex labels to uniquely label all edges.

The first step in Algorithm 7 is similar to Algorithm 5. A BFS is performed in order to determine the level $L(v)$

Algorithm 8 Truncated BFS subroutine in BFS-BiCC to identify articulation points and track component vertex set.

```
1:  procedure BFS-LV(G(V, E), L, v, u, visited)
2:      Insert u into Q
3:      Insert u into V_u
4:      visited(u) ← true
5:      visited(v) ← true
6:      vid_low ← u
7:      while Q ≠ ∅ do
8:          for all x ∈ Q do
9:              Remove x from Q
10:             for all ⟨w, x⟩ ∈ E where visited(w) = false do
11:                 if L(w) < L(u) then
12:                     return (L(w), ∅, ∅)
13:                 else
14:                     Insert w into Q
15:                     Insert w into V_u
16:                     visited(w) ← true
17:                     if w < vid_low then
18:                         vid_low ← w
19:     return (L(u), vid_low, V_u)
```

and parent $P(v)$ for each vertex v in V. Additionally, we store the output of the level-synchronous BFS as a list of queues LQ. Each $Q_i \in LQ$ contains all the vertices at a distance i from the root vertex. We inspect the queues in reverse order, from maximum level m through level 1 (level 0 is considered as the level containing the root). If u has already been assigned a Par value, we know that the biconnected component that contains u has already been discovered via another child of $P(u)$. Otherwise, we perform a similar BFS as in Algorithm 5. However, in this search, we also track the lowest vertex identifier encountered, vid_{low}, as well as a list of all unique vertices encountered, V_u.

If we determine that v is an articulation vertex based on the retrieved minimum level l, we then proceed to label all Par and Low of all encountered vertices w. $Par(w)$ is set to point to the BiCC parent articulation vertex of v, while $Low(w)$ is set to vid_{low}. This ensures a unique and consistent labeling across vertices within the components. Note that it is not necessary to create the Par values, since simply tracking the articulation vertices is sufficient to correctly label edges. We choose a Par array as opposed to a boolean array or vertex list in order to be consistent with the output of prior BiCC algorithms.

Once a component is identified, V_u, which is the discovered component minus the articulation vertex, is considered removed from G (line 21 of Algorithm 7). We do not modify G. Instead, we maintain $valid$, a shared boolean array of size n that signifies the current state of a vertex. When we remove a v from G, we set $valid(v) = false$. Because we are working from the highest-level leaves of the tree to the root in T, we can safely do this. On any given level i, all articulation vertices on level $i-1$ will be discovered. This is because separate biconnected components cannot exist through articulation vertices by Corollary III.3, and all such articulation vertices will be discovered by Proposition III.1. It is also guaranteed that no vertices in the biconnected component will have been previously removed, as there is no way they could have been encountered by a successful articulation vertex search.

The final step is to label all edges between all u and v. Although it is possible to do this in the inner BFS loop, it is simpler and more cache-friendly to do the separate step. For labeling, if both vertices have the same Low value, we know they were discovered during the same search, and therefore the edge is contained in their component. If the edge is between a child and its parent articulation point, we apply the Low label of the child. We don't further explicitly label vertices as belonging to a component, since articulation vertices exist in multiple components, and the information is readily retrievable by examining Low and Par arrays.

2) Parallelization: Our primary avenue for parallelization is across all vertices in the current queue level (step 9). We could also parallelize the BFS-LV search. However, in our implementation, the BFS-LV is only parallelized once we reach level 1, since we observe that in in most real-world graphs that have a massive biconnected component, a randomly-selected root vertex is likely to be contained in the giant component. We do not require additional synchronization while updating the Low and Par values. Should the parent be an articulation vertex and the children be contained in the same component, the same Par and Low values will subsequently be given to all vertices in the component. Thus, all the concurrent writes are benign races.

3) Algorithm Analysis: The dominant step in the algorithm is the number of invocations of BFS-LV and the cumulative number of edge inspections performed through BFS-LV, with the naïve approach requiring an upper bound of $O(nm)$ work. The rest of the steps (initialization, initial BFS, Low and Par labeling) require $\theta(n+m)$ work. The naïve approach to determine articulation points is somewhat inefficient, since there is no ordering imposed on invocations of BFS-L. However, in the full algorithm, we inspect vertices in level-synchronous order, and once a biconnected component is identified, all visited vertices are marked as invalid. Thus, there are no further unnecessary traversals. We also truncate BFS-LV as soon as we encounter a vertex at the level of the parent. We observe that the cumulative number of edge examinations is a small constant multiplicative factor of the total number of edges, and so the work performed is linear in practice. The level-synchronous approach of examining the vertices implies that the parallel time would be proportional to the the graph diameter.

B. Coloring-based BiCC Method

Instead of potentially performing a full BFS from each vertex in T, it is also possible to compute the same Par and Low values using the *color propagation* technique. Color propagation is an iterative strategy that is similar to recursive doubling, and we have previously used it to develop practical parallel algorithms for connected components in undirected graphs, as well as the weakly and strongly connected components in directed graphs [29].

We define the *lowest common ancestor* (LCA) p of any two neighboring vertices u and v in a BFS tree $T\langle L, P\rangle$ to be

the lowest-level vertex that both vertices share some ancestral relationship with in T. Should a parent-child relationship exist between these two vertices, $P(v) = u$ or $P(u) = v$, then the lowest common ancestor is simply the parent vertex.

Our Coloring-based approach is based on the following observation for biconnected components with at least three vertices.

Proposition III.4. *In a biconnected component with at least three vertices, determine the LCA of all pairs of neighboring vertices. At least two vertices in the component will have their lowest-level LCA set to the parent articulation point.*

Proof: Any biconnected component containing at least three vertices has the requirement that the articulation vertex has at least two children. If the articulation vertex has only one child, then the removal of this child would disconnect the component and the component is therefore not biconnected. Additionally, the lowest-level LCA for each of the these two child vertices will always be articulation vertex, as per Proposition III.1. It should also be noted that for any biconnected component of size larger than three, it is highly likely that two neighboring vertices of higher levels connected through an edge not in T have each of their lowest level mutual parents set as the parent articulation vertex as well. ■

1) Identifying BiCC with Color Propagation: Our approach for determining biconnected components is given by Algorithm 9. This algorithm once again determines the same Par and Low values as the BFS approach. However, instead of propagating the values to vertices within the same BiCC through a search, we simply propagate them to their neighbors one iteration at a time under certain constraints.

The initialization steps are the same as before. We select a root and perform the BFS to create the parent and level arrays. We use this information to initialize the Par values for each vertex v to the lowest-level LCA among it and all of its neighbors $\langle u, v\rangle \in E$. We omit pseudocode for LCA, as it is a well-known algorithm.

Once initialization is complete, we begin our primary coloring loop. The goal of coloring is to color all vertices in a biconnected component, $v \in B$ under a parent articulation vertex of p, with $Par(v) = p$. Additionally, we want to color all $Low(v)$ as the vertex in B with the lowest vertex identifier.

A Par value is propagated from a vertex v to a neighbor u if the level of $Par(v)$ is less than the level of $Par(u)$. To ensure that no Par value is passed down from an articulation vertex to its child, we don't propagate Par values from a parent to a child unless the Par value of the child is not equal to the parent (i.e., we know there is a path from the child to a vertex of lower level than the parent, so by Proposition III.1, we know the parent is in the same biconnected component as its child). We only propagate Low values between vertices that have the same Par value, as the same Par value already indicates that they are in the biconnected component.

We know that in any non-trivial biconnected component there must be at least two vertices with their Par value initialized to the parent articulation vertex for the component.

Algorithm 9 Color propagation-based algorithm to perform BiCC decomposition.

```
1:  procedure COLOR-BICC(G(V, E))
2:      for all v ∈ V do
3:          Art(v) ← false
4:          Low(v) ← v
5:          Par(v) ← v
6:          Insert v into Q
7:          queued ← true
8:      Select a root vertex r
9:      P, L ← BFS(G, r)
10:     Init-LCA(G, P, L, Par)
11:     while Q ≠ ∅ do
12:         for all v ∈ Q do
13:             Remove v from Q
14:             for all ⟨u, v⟩ ∈ E do
15:                 if Par(u) = v then
16:                     continue
17:                 if L(Par(v)) > L(Par(u)) then
18:                     Par(u) ← Par(v)
19:                     if queued(u) = false then
20:                         queued(u) ← true
21:                         Insert u into Q
22:                 if Par(v) = Par(u) then
23:                     if Low(v) < Low(u) then
24:                         Low(u) ← Low(v)
25:                         if queued(u) = false then
26:                             queued(u) ← true
27:                             Insert u into Q
28:             if any u got queued and queued(v) = false then
29:                 queued(v) ← true
30:                 Insert v into Q
31:         for all v ∈ Q do
32:             queued(u) ← false
33:     for all e = ⟨u, v⟩ ∈ E do
34:         if Low(v) = Low(u) or Par(u) = v then
35:             BiCC(e) ← low(u)
36:         else
37:             BiCC(e) ← low(v)
```

Algorithm 10 Initialize the LCA for all neighbors using parents and level information.

```
1: procedure INIT-LCA(G(V, E), P, L, Par)
2:     for all v ∈ V do
3:         for all ⟨u, v⟩ ∈ E do
4:             w = LCA(v, u, G, P, L)
5:             if L(w) ≤ L(Par(v)) then
6:                 Par(v) = w
```

These lowest-level Par values will freely propagate to all vertices in the biconnected component, with the exception of vertices not initialized to a Par besides their parents, which are vertices that have no immediate non-tree connection to a vertex of a lower level. However, these vertices do have a path to their grandparent vertex (which may be the parent articulation vertex) through either a vertex at the same level, non-tree edge (which also must be a child of the same parent) or one of their children. Using a recursive argument along this path, the directional restriction we have on Par value propagation will eventually be lifted as this lower level Par will finally reach the original vertex. When this happens, the lowest-level parent articulation vertex Par will be able to reach this vertex as well through any path. We also know that if the Par value reaches all vertices in the component, then so must the correct Low value, as it is unique and will begin propagating immediately when the vertex which has the Low vertex identifier value gets their Par set, or initialized

to the final correct value.

We rely on a queue Q to avoid having to examine every vertex at every iteration. We further rely on a *pushing* as opposed to *pulling* form of coloring. While visiting vertex v, we examine and overwrite the colors of all its neighbors, u. With a pulling methodology, we would only overwrite the color of v with the best color from all of u. We empirically find pushing to be faster. We mitigate the race condition created by two vertices both overwriting the color of u by adding both v and u to the queue. If v had attempted to push the superior color but got it overwritten, on the next iteration v will attempt to push it again and succeed.

2) Parallelization: As with the BFS-BiCC algorithm, parallelization of the first stage of the algorithm is straightforward as it utilizes a standard BFS. Parallelization of the coloring stage is a bit more involved, but still relatively simple to implement. We parallelize over the queue in step 12, with each thread examining and propagating colors from a limited subset of vertices. To avoid the overhead associated with writing to a shared queue, we instead have each thread place their vertices in a thead-owned queue. Once a thread completes its iteration, its queue is copied into the global queue for the next iteration. As these operations are non-blocking, there is minimal overhead. Like the previous approach, the final labeling step over all edges is easy to parallelize.

3) Algorithm Analysis: The key step in Color-BiCC is the coloring phase, and the work performed depends on the number of times a vertex is inserted into the queue and cumulative number of edges inspected. The rest of the steps can be performed in $\theta(n + m)$ work. The upper bound on work for coloring-based connected components and strongly connected components algorithms is $O(n^2)$ [25], but the observed performance is linear in the number of edges for real-world, low-diameter graph instances [6], [29]. Because we impose additional constraints in this case for BiCC and use precomputed LCA information to direct the color propagation, the work performed is input-dependent. We quantify the overhead of coloring through the additional edge inspections required (a multiplicative factor) over the baseline value of m. We report this value for all the test instances in the next section. Note that concurrency depends on the size of the queue for each iteration, and is not dependent on graph diameter.

IV. Experimental Setup

Experiments were performed on a dual-socket Intel system with 64 GB main memory and Xeon E5-2670 (Sandy Bridge) CPUs clocked at 2.60 GHz, and each having 20 MB last-level cache. OpenMP was used for parallelization. Several large real-world graphs were used in our study. These are listed in Table I. These graphs were retrieved from a number of sources, namely the SNAP database [20], the Koblenz Network Collection [19], the 10th DIMACS Implementation Challenge [5], and the University of Florida Sparse Matrix Collection [12]. The R-MAT [9] and $G(n, p)$ (GNP) networks were generated with the GTGraph [21] suite using the default parameters. Only the largest connected component for each graph was taken. Directed edges were considered undirected. Multiple edges and self loops were removed. This was done to reduce noise in results and simplify analysis and comparison between tested algorithms.

TABLE I
Network sizes and parameters for all networks. The columns are #vertices, #edges, average and max-degree, approximate diameter, # of BiCCs and size of the largest BiCC.

Network	n	m	d_{avg}	d_{max}	$\widetilde{dia}$	# Bi	Max Bi
LiveJournal	4.8M	43M	18	20K	21	1.1M	3.7M
Orkut	3.1M	117M	76	33K	11	68K	3.0M
WikiLinks	26.0M	543M	42	4.3M	86	3.5M	22M
ItWeb	41.0M	1.0B	50	1.3M	46	5.0M	33M
Friendster	63.0M	1.6B	53	5.2K	34	13M	49M
Cube	2.1M	59M	56	67	157	1	2.1M
Kron_21	1.5M	91M	118	214K	8	238K	1.3M
R-MAT_24	7.7M	133M	35	257K	11	2.2M	5.4M
GNP_1	10.0M	200M	40	152	7	1	10M
GNP_10	10.0M	200M	40	80	19	19	5M

Friendster, LiveJournal, and Orkut are social networks [34], [3], [8]. WikiLinks is the cross-link network between articles on Wikipedia [33]. Cube is 3D coupled consolidation problem of a cube discretized with tetrahedral finite elements [18]. R-MAT_24 is an R-MAT graph of scale 24. The Kron_21 graph is a scale 21 graph created from the Kronecker generator of the Graph500 benchmark [15]. Finally, GNP_1 and GNP_10 refer to Erdős-Rényi random $G(n, p)$ graphs with 1 and 10 large biconnected components, respectively. The GNP_10 network was created to have 10 large biconnected components connected through 9 bridges. The components were generated independently with geometrically decreasing sizes (5.0M, 2.5M, 1.25M, ...) with the bridges added manually by connecting the independent components via single edges.

V. Results

We compare the performance of our two new algorithms to Cong and Bader's improvement to the Tarjan-Vishkin algorithm, as well as the Hopcoft-Tarjan serial algorithm. We will demonstrate absolute speedups relative to prior work, strong scaling of our algorithms, an analysis of each of our algorithms with respect to the basic graph computations on which they are based on, as well as a more general analysis of the biconnected component size and count distribution between real world and synthetic graphs.

A. Execution times and Scaling

Table II gives the serial execution time of the Hopcroft-Tarjan algorithm, the parallel times of our coloring and BFS algorithms with 32 threads, the TV-Filter implementation run with 32 threads, as well as our best speedup over TV-Filter. All reported times are the averages over five independent tests. We select the highest out-degree vertex as the root for reasons that will be noted. The fastest time over all approaches is given in bold. As previously mentioned, memory limitations for the

TABLE II
EXECUTION TIME (SECONDS) RESULT COMPARISON BETWEEN THE SERIAL HOPCROFT-TARJAN ALGORITHM, TV-FILTER ALGORITHM ON 32 THREADS, AND THE NEW BFS-BICC AND COLOR-BICC APPROACHES ON 32 THREADS.

Network	HT	TV-Filter	BFS-BiCC	Color-BiCC	Speedup vs. TV-F
LiveJournal	2.1	1.6	**0.38**	0.61	4.2
Orkut	3.4	1.8	**0.49**	0.93	3.6
WikiLinks	25	24	**7.0**	20	3.4
ItWeb	19	–	50	**3.3**	–
Friendster	79	–	**20**	48	–
Cube	1.2	0.64	**0.17**	0.51	3.8
Kron_21	1.8	2.3	**0.60**	2.2	3.8
R-MAT_24	4.7	5.8	**1.5**	5.6	3.9
GNP_1	11	5.8	5.0	**4.8**	1.2
GNP_10	6.5	5.9	12	**4.0**	1.5

TABLE III
EXECUTION TIME (IN SECONDS) COMPARISON BETWEEN THE SERIAL ALGORITHM, A STANDARD BFS RUN, AND THE BFS-BICC ALGORITHM. ADDITIONALLY, A RATIO OF THE AVERAGE NUMBER OF EDGES EXAMINED DURING THE INNER-LOOP BFS IS GIVEN.

Network	HT	BFS	(1)	(16)	(32)	Edge Ratio
LiveJournal	2.1	0.032	2.5	0.56	0.38	1.0
Orkut	3.4	0.025	3.3	0.75	0.49	0.60
WikiLinks	25	0.40	29	9.3	7.0	1.3
ItWeb	19	0.41	460	70	50	1.6
Friendster	79	0.46	150	33	20	0.95
Cube	1.2	0.042	0.98	0.20	0.17	0.031
Kron_21	1.8	0.030	1.1	0.71	0.60	0.015
R-MAT_24	4.7	0.12	6.2	2.0	1.5	0.042
GNP_1	11	0.082	37	10	5.0	3.1
GNP_10	6.5	0.15	31	16	12	19

TV-Filter algorithm prevented us from obtaining time results on the two largest graphs, ItWeb and Friendster.

From Table II, it is apparent that the BFS-based algorithm demonstrates the fastest absolute execution times for a majority of test cases. Our coloring algorithm demonstrates the best performance on the remaining test cases. Both of our algorithms demonstrate considerable speedup over both the serial algorithm as well as the TV-Filter algorithm. As will be explained, the relative performance benefit between the BFS and coloring algorithms is highly dependent on graph structure.

Figure 1 gives the strong scaling of the BFS, coloring, and TV-Filter algorithms from 1 to 32 threads on all 10 test graphs. Reported speedups are relative to the execution time of Hopcroft-Tarjan. Similar results as Table II can be observed from Figure 1, with the BFS algorithm demonstrating the fastest execution time and best speedups on a majority of tests.

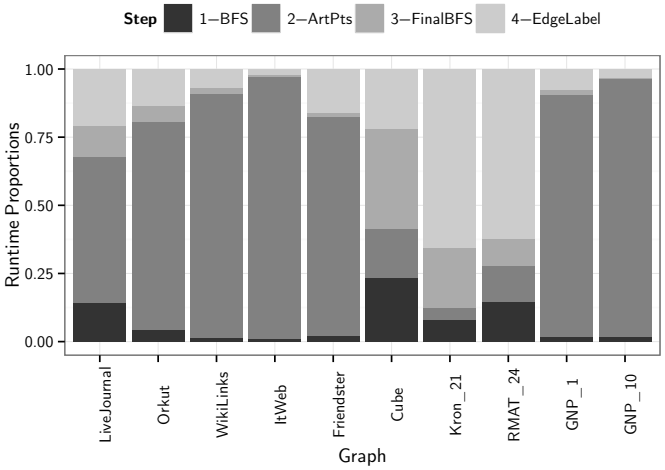

Fig. 2. Per-step execution time breakdown of the BiCC-BFS approach.

B. Breadth-First Search Analysis

Table III gives the execution time of the BFS-BiCC algorithm with 1, 16, and 32 threads. Additionally, the running time of just a single BFS exploration of the entire graph is given along with the serial algorithm for comparison. We also report a work estimate termed *edge ratio* in the final column. The edge ratio is the total number of edges explored during the second stage of the BFS-BiCC algorithm (where articulation vertices are being searched for) divided by the total number of edges in the graph. This essentially gives the multiplicative factor of additional/lesser work being performed in comparison to the Hopcroft-Tarjan algorithm, where all m edges are necessarily examined.

As can be observed, the graphs that have low edge ratios demonstrate the best performance with BFS-BiCC relative to the serial approach. In fact, for all four networks where the edge ratio is considerably less than 1.0, such as Orkut, Cube, Kron, and R-MAT, single thread execution time for BFS-BiCC is faster than the time for Hopcroft-Tarjan. As explained before, this is because of the direction-optimizing BFS that is very effective for certain low-diameter graphs.

Figure 2 gives the ratio of time spent in each of the four stages of the BFS-BiCC algorithm (initial BFS tree creation, articulation point identification, final level BFS run(s), edge labeling). For most graphs, the second stage dominates the overall execution time, as expected. The exception to this is a few of the graphs mentioned previously that have a low edge ratio. For those, the necessary $O(m)$ work required by edge labeling is the largest portion of execution time.

C. Color Propagation Analysis

We will now look at similar performance metrics obtained for the Color-BiCC algorithm. Table IV gives the execution time of Color-BiCC with 1, 16, and 32 threads, as well as the execution time of a color propagation algorithm for identifying connected components in a graph.

We also compute an edge ratio metric in this case. It is calculated as the total number of edge propagations divided by the total number of edges. There is a moderate correlation between the edge ratio and performance of the coloring algorithm. This is especially apparent for the Kron_21 and R-MAT graphs, which demonstrate the highest edge ratio, and correspondingly are the only networks where fully-parallel coloring offers no advantage over the serial algorithm. It is noted that this ratio does not account for the initialization of the LCAs, which does make a considerable contribution to the

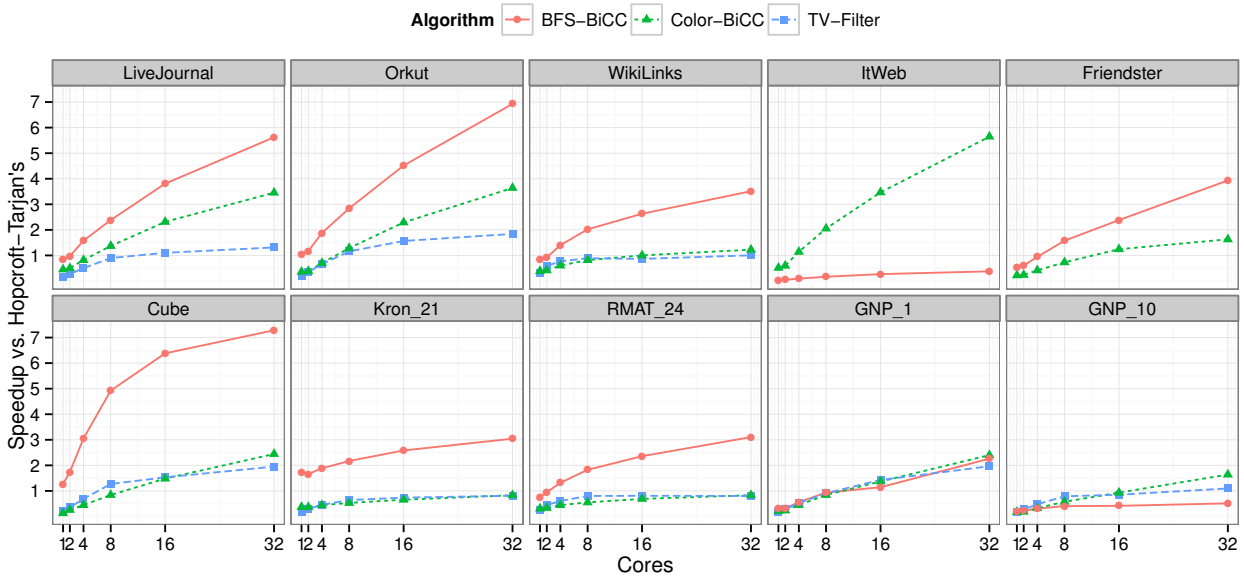

Fig. 1. Parallel scaling of BFS and Coloring approaches as well as Cong and Bader's implementation relative to the serial Hopcroft-Tarjan algorithm.

TABLE IV
EXECUTION TIME (IN SECONDS) COMPARISON BETWEEN THE SERIAL ALGORITHM, A COLOR PROPAGATION ALGORITHM FOR CONNECTED COMPONENTS, AND THE COLOR-BICC ALGORITHM. ADDITIONALLY, THE TOTAL NUMBER OF COLOR PROPAGATIONS DIVIDED BY THE NUMBER OF EDGES IN THE NETWORK IS REPORTED.

Network	HT	Color	(1)	(16)	(32)	Edge Ratio
LiveJournal	2.1	0.48	4.7	0.92	0.61	0.10
Orkut	3.4	0.44	9.4	1.5	0.93	0.063
WikiLinks	25	4.6	63	24	20	0.30
ItWeb	19	–	36	5.4	3.3	0.083
Friendster	79	21	360	63	48	0.063
Cube	1.2	0.19	9.7	0.84	0.51	0.17
Kron_21	1.8	0.43	5.0	2.8	2.2	0.62
R-MAT_24	4.7	1.2	16	6.8	5.6	0.57
GNP_1	11	1.8	53	8.3	4.8	0.064
GNP_10	6.5	2.0	37	7.0	4.0	0.062

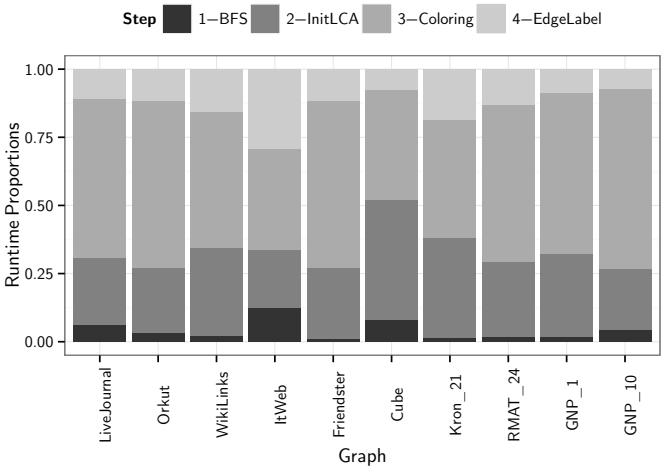

Fig. 3. Per-step breakdown of the Coloring approach.

overall time.

Figure 3 gives a per-step breakdown as Figure 2, with the four steps as the initial BFS tree creation, the initialization of the LCAs for all vertices, the primary coloring stage, and the final edge labeling. Compared to the breakdown for the BFS algorithm, the coloring algorithm has much more consistent ratios of times spent in all four stages across all networks. As is also observed, a majority of time is spent in the LCA initialization and coloring stages, with the coloring stage taking approximately twice as much time as the initialization stage.

D. Performance impact of root vertex choice

For the breadth-first search algorithm, the vertex selected as the initial root can considerably impact the overall running time. Table V demonstrates this difference. By selecting the vertex with the highest out-degree, the work required during the inner loop of the BFS algorithm can be minimized. This is because the final large biconnected component is usually found by the first vertex encountered in the final level queue. For networks containing a node with an exceptionally large out degree, such as WikiLinks (4.3M), the resultant difference in running time can be quite considerable. As the effort to find this vertex is minimal and often tracked during graph creation, the extra work required for this simple heuristic is minimal with regards to the possible payoff.

For coloring algorithms, there is minimal correlation between the initial root vertex and the execution time of the algorithm. Ideally, the selected root should minimize the number of traversals required to initialize the LCA for all vertices, as well as minimize the number of the color propagations required. Selecting such a vertex seems like a challenging problem to be solved and the solution with the highest degree heuristic does not seem satisfactory.

TABLE V
SPEEDUPS RESULTING FOR BOTH THE BFS AND COLORING ALGORITHMS WITH THE HEURISTICALLY-CHOSEN ROOT VERTEX COMPARED TO THE AVERAGE RESULT OVER 20 RANDOMLY SELECTED ROOT VERTICES.

Network	BFS-BiCC Speedup	BFS-Color Speedup
LiveJournal	2.7	0.98
Orkut	2.7	0.94
WikiLinks	930	1.1
ItWeb	1.8	0.97
Friendster	3.7	1.0
Cube	0.92	0.98
Kron_21	3.5	1.1
R-MAT_24	12	0.96
GNP_1	1.1	1.0
GNP_10	0.42	1.0

VI. CONCLUSIONS

This paper introduces two novel shared-memory parallel algorithms for finding the biconnected components of an undirected graph. Since they both use simple and well-known subroutines, practical and efficient parallel implementations are much more feasible compared to prior algorithms. Additionally, our implementations of both of these algorithms offer considerable speedup over the TV-Filter implementation, and are more memory-efficient. We will extend the theoretical analysis of these algorithms by identifying worst-case instances for the BiCC-BFS approach. We will also focus on performance tuning and experimental analysis of these algorithms in future work. Using the BiCC decomposition and relative component sizes to characterize graph structure, particularly community structure in real-world social networks, will perhaps provide novel insight.

VII. ACKNOWLEDGMENTS

This work is supported by NSF grant ACI-1253881. We thank Siva Rajamanickam (Sandia Labs) for helpful discussions and guidance, and for facilitating timely access to computing resources at Sandia. Sandia National Laboratories is a multi-program laboratory managed and operated by Sandia Corporation, a wholly owned subsidiary of Lockheed Martin Corporation, for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-AC04-94AL85000.

REFERENCES

[1] G. Ausiello, D. Firmani, et al. Real-time monitoring of undirected networks: Articulation points, bridges, and connected and biconnected components. *Networks*, 59(3):275–288, 2012.

[2] G. Ausiello, D. Firmani, L. Laura, and E. Paracone. Large-scale graph biconnectivity in MapReduce. *Department of Computer and System Sciences Antonio Ruberti Technical Reports*, 4(4), 2012.

[3] L. Backstrom, D. Huttenlocher, J. Kleinberg, and X. Lan. Group formation in large social networks: Membership, growth, and evolution. In *Proc. Conf. on Knowledge Discovery and Data Mining (KDD)*, 2006.

[4] D. A. Bader and J. JáJá. SIMPLE: A methodology for programming high performance algorithms on clusters of symmetric multiprocessors (SMPs). *Journal of Parallel and Distributed Computing*, 58(1):92–108, 1999.

[5] D. A. Bader, H. Meyerhenke, P. Sanders, and D. Wagner. Graph partitioning and graph clustering, 10th DIMACS implementation challenge workshop. *Contemporary Mathematics*, 588, 2013.

[6] J. Barnat and P. Moravec. Parallel algorithms for finding SCCs in implicitly given graphs. *Formal Methods: Applications and Technology*, 4346:316–330, 2006.

[7] S. Beamer, K. Asanović, and D. Patterson. Direction-optimizing breadth-first search. In *Proc. Supercomputing (SC)*, 2012.

[8] M. Cha, H. Haddadi, F. Benevenuto, and K. P. Gummadi. Measuring user influence in Twitter: The million follower fallacy. In *Proc. Int'l. Conf. on Weblogs and Social Media (ICWSM)*, May 2010.

[9] D. Chakrabarti, Y. Zhan, and C. Faloutsos. R-MAT: A recursive model for graph mining. In *Proc. SIAM Int'l. Conf. on Data Mining (SDM)*, 2004.

[10] G. Cong and D. A. Bader. TV-Filter biconnected components implementation, 2004. http://www.cc.gatech.edu/~bader/code.html, last accessed Aug 25, 2014.

[11] G. Cong and D. A. Bader. An experimental study of parallel biconnected components algorithms on symmetric multiprocessors (SMPs). In *Proc. Int'l. Parallel and Distributed Processing Symp. (IPDPS)*, 2005.

[12] T. A. Davis and Y. Hu. The University of Florida sparse matrix collection. *ACM Transactions on Mathematical Software*, 38(1):1–25, 2011.

[13] D. M. Eckstein. BFS and biconnectivity. Technical Report 79-11, Dept. of Computer Science, Iowa State University of Science and Technology, 1979.

[14] J. A. Edwards and U. Vishkin. Better speedups using simpler parallel programming for graph connectivity and biconnectivity. In *Proc. Int'l. Workshop on Programming Models and Applications for Multicores and Manycores (PMAA)*, 2012.

[15] Graph 500. http://www.graph500.org, last accessed Aug 25, 2014.

[16] J. Hopcroft and R. Tarjan. Efficient algorithms for graph manipulation. *CACM*, 16(6):374–378, 1973.

[17] T.-S. Hsu, V. Ramachandran, and N. Dean. Implementation of parallel graph algorithms on the MasPar. In N. Dean and G. E. Shannon, editors, *Third DIMACS Implementation Challenge: Parallel Algorithms*, volume 15 of *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, pages 165–198. AMS, 1994.

[18] C. Janna, M. Ferronato, and G. Gambolati. Parallel inexact constraint preconditioning for ill-conditioned consolidation problems. *Computational Geosciences*, 16(3):661–675, 2012.

[19] J. Kunegis. KONECT - the Koblenz network collection. konect.uni-koblenz.de, last accessed Aug 25, 2014.

[20] J. Leskovec. SNAP: Stanford network analysis project. http://snap.stanford.edu/index.html, last accessed Aug 25, 2014.

[21] K. Madduri and D. A. Bader. GTgraph: A suite of synthetic graph generators. http://www.cse.psu.edu/~madduri/software/GTgraph/, last accessed Aug 25, 2014.

[22] Y. Maon, B. Schieber, and U. Vishkin. Parallel ear decomposition search (eds) and st-numbering in graphs. *Theoretical Computer Science*, 47:277–298, 1986.

[23] G. L. Miller and V. Ramachandran. A new graph triconnectivity algorithm and its parallelization. *Combinatorica*, 12(1):53–76, 1992.

[24] H. Nagamochi and T. Ibaraki. A linear-time algorithm for finding a sparse k-connected spanning subgraph of a k-connected graph. *Algorithmica*, 7(1-6):583–596, 1992.

[25] S. Orzan. *On Distributed Verification and Verified Distribution*. PhD thesis, Vrije Universiteit, 2004.

[26] C. A. R. Pinheiro. *Social Network Analysis in Telecommunications*. SAS Institute Inc, 2011.

[27] A. E. Sarıyüce, K. Kaya, E. Saule, and U. V. Catalyürek. Incremental algorithms for closeness centrality. In *IEEE International Conference on BigData*, 2013.

[28] C. Savage and J. JáJá. Fast, efficient parallel algorithms for some graph problems. *SIAM J. Computing*, 10(4):682–691, 1981.

[29] G. M. Slota, S. Rajamanickam, and K. Madduri. BFS and Coloring-based parallel algorithms for strongly connected components and related problems. In *Proc. Int'l. Parallel and Distributed Processing Symp. (IPDPS)*, 2014.

[30] R. E. Tarjan and U. Vishkin. An efficient parallel biconnectivity algorithm. *SIAM Journal on Computing*, 14(4):862–874, 1985.

[31] Y. H. Tsin and F. Y. Chin. Efficient parallel algorithms for a class of graph theoretic problems. *SIAM J. Computing*, 31(2):245–281, 1984.

[32] J. Westbrook and R. E. Tarjan. Maintaining bridge-connected and biconnected components on-line. *Algorithmica*, 7(1-6):433–464, 1992.

[33] Wikimedia Foundation. Wikipedia links, english network dataset – KONECT, Oct. 2013.